Parallel & Distributed Computing

UCS645

Assignment - 1

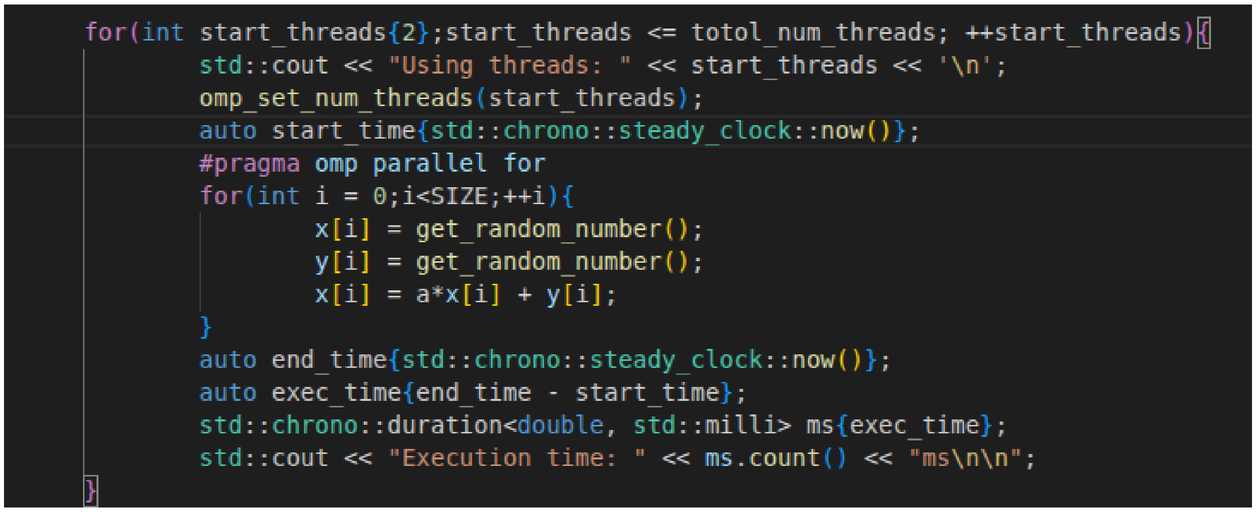
Name - Kartik Goel
Roll No. - 102303182

System Environment: Linux (Ubuntu), GCC Compiler with OpenMP Support

AIM - To learn OpenMP basics

Question 1 - DAXPY Loop

Using code:-



```
for(int start_threads{2};start_threads <= totol_num_threads; ++start_threads){
        std::cout << "Using threads: " << start_threads << '\n';
        omp_set_num_threads(start_threads);
        auto start_time{std::chrono::steady_clock::now()};
        #pragma omp parallel for
        for(int i = 0;i<SIZE;++i){
                x[i] = get_random_number();
                y[i] = get_random_number();
                x[i] = a*x[i] + y[i];
        }
        auto end_time{std::chrono::steady_clock::now()};
        auto exec_time{end_time - start_time};
        std::chrono::duration<double, std::milli> ms{exec_time};
        std::cout << "Execution time: " << ms.count() << "ms\n\n";
}
```

The task was to see the speedup in terms of execution time as we increase the number of threads, and in this question each iteration of the for loop is independent as the result of one iteration does not depend on the other iteration's result.

ON EXECUTION:-

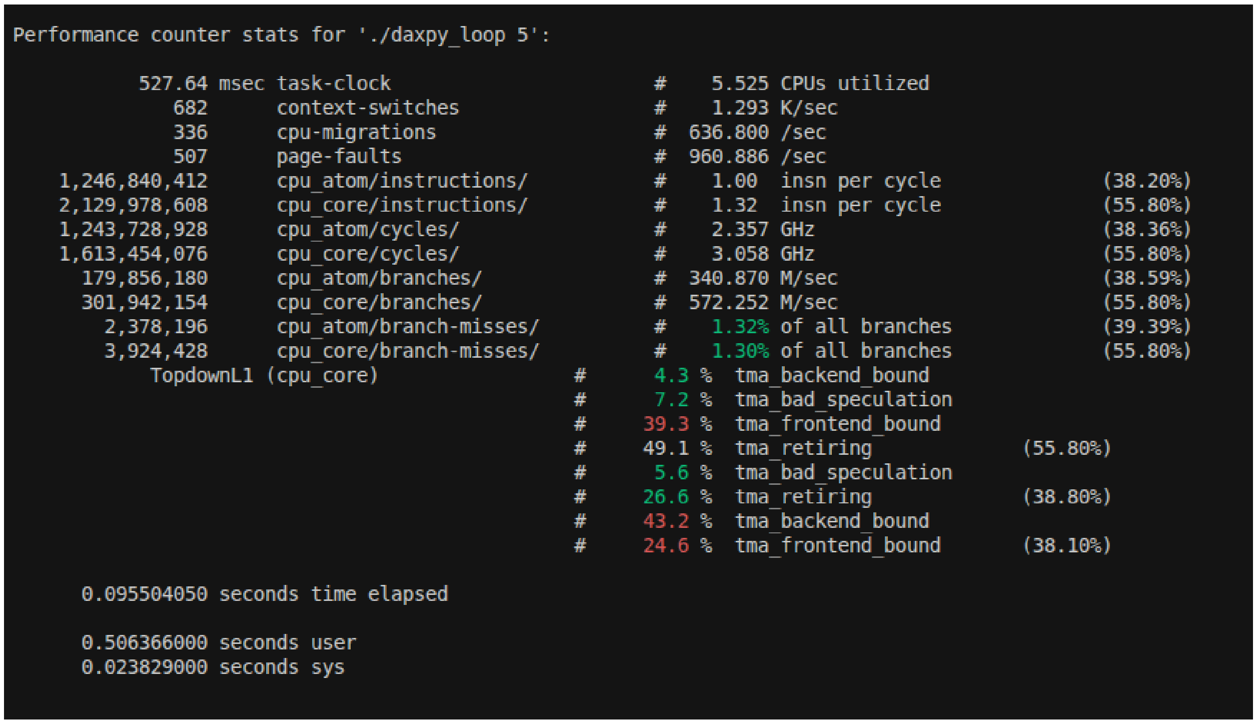
Threads	**Execution Time (ms)**	**Observation**
2	12.1507	Baseline
...	...	Decreasing in general but not in order (explained in inference)
22	1.02988	Min Execution Time
24	1.22391	Execution time increases again

INFERENCE:-

After executing this program we can infer that execution time decreases very significantly as we increase the number of threads or we can also say that there is a huge increase in the execution speed of the program.
Repeated executions of the program showed a general trend of decreasing execution time as the number of threads increased from 2 up to a maximum of 20–23 (22 in this specific test). However, this decrease was not strictly monotonic; for example, the execution time with 14 threads might occasionally be slightly higher than with 13 threads.
But yes the execution time increases after a particular point because of the following possible reasons
1. When more than a particular number of threads are used the context switching time (because the CPU has to give each thread equal amount of time it switches between threads this is context switching) becomes significant as a result a context switch takes valuable CPU cycles that do not contribute to the actual computation.

2. Core local memory (L1/L2 Cache) is small and fast. When many threads access distinct parts of large vectors (x and y), they repeatedly displace each other's data, causing **Cache Misses**. This forces the CPU to wait for slow RAM access.
3. Simultaneous access to shared resources (such as memory, locks, disk I/O, or caches) by multiple threads necessitates that these threads wait for one another.
4.When system memory (RAM) is depleted due to high memory usage, the system resorts to using much slower disk-based virtual memory (swapping). This process is significantly slower than using physical memory.

OBSERVATION FROM PERF STAT:



```
Performance counter stats for './daxpy_loop 5':

           527.64 msec task-clock                       #    5.525 CPUs utilized
              682      context-switches                 #    1.293 K/sec
              336      cpu-migrations                   #  636.800 /sec
              507      page-faults                      #  960.886 /sec
    1,246,840,412      cpu_atom/instructions/           #    1.00  insn per cycle              (38.20%)
    2,129,978,608      cpu_core/instructions/           #    1.32  insn per cycle              (55.80%)
    1,243,728,928      cpu_atom/cycles/                 #    2.357 GHz                         (38.36%)
    1,613,454,076      cpu_core/cycles/                 #    3.058 GHz                         (55.80%)
      179,856,180      cpu_atom/branches/               #  340.870 M/sec                       (38.59%)
      301,942,154      cpu_core/branches/               #  572.252 M/sec                       (55.80%)
        2,378,196      cpu_atom/branch-misses/          #    1.32% of all branches             (39.39%)
        3,924,428      cpu_core/branch-misses/          #    1.30% of all branches             (55.80%)
            TopdownL1 (cpu_core)                 #      4.3 %  tma_backend_bound
                                                 #      7.2 %  tma_bad_speculation
                                                 #     39.3 %  tma_frontend_bound
                                                 #     49.1 %  tma_retiring             (55.80%)
                                                 #      5.6 %  tma_bad_speculation
                                                 #     26.6 %  tma_retiring             (38.80%)
                                                 #     43.2 %  tma_backend_bound
                                                 #     24.6 %  tma_frontend_bound       (38.10%)

      0.095504050 seconds time elapsed

      0.506366000 seconds user
      0.023829000 seconds sys
```

1. Resource Utilization:-

Parallel Efficiency: With 5.525 CPUs utilized, the system is successfully running your threads in parallel across multiple cores.

Hybrid Execution: The workload is split between Performance-cores (running at 3.058 GHz) and Efficiency-cores (running at 2.357 GHz).

2. Instruction Efficiency:-

P-Core Performance: The Performance cores achieve an IPC (Instructions Per Cycle) of 1.32, showing relatively healthy throughput for this math-heavy task.

E-Core Lag: The Efficiency (Atom) cores are slower, with an IPC of 1.00, meaning they process fewer instructions per clock tick.

3. Primary Bottlenecks:-

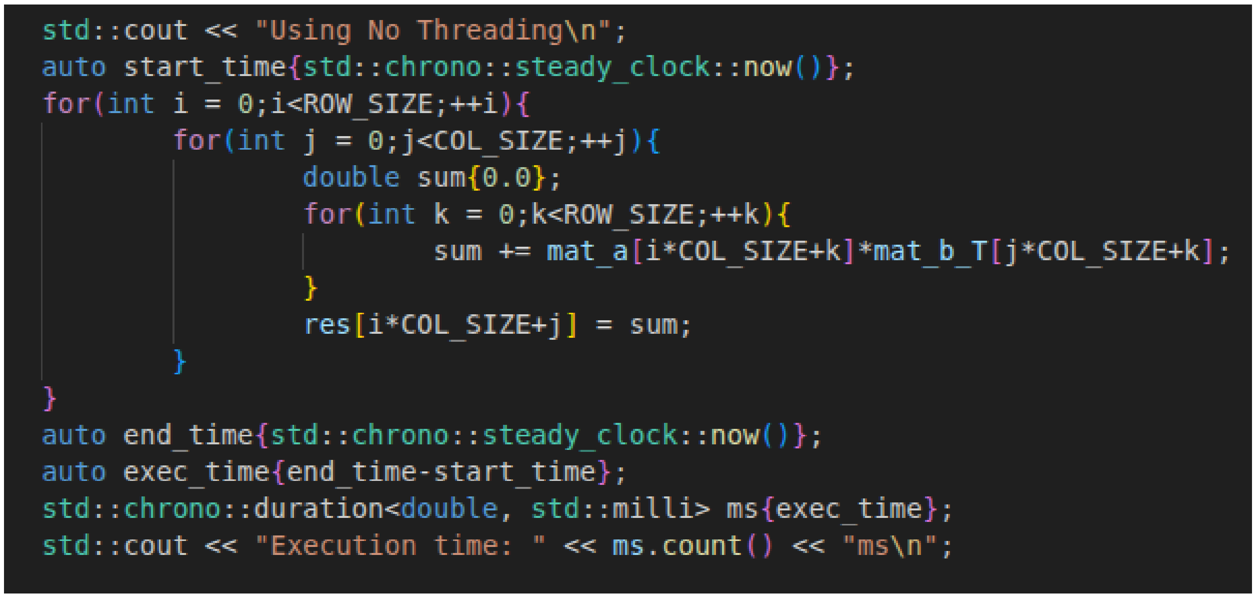
Frontend Bound (39.3%): This is your biggest bottleneck; the CPU is often waiting for instructions to be fetched or decoded before it can even start the math.

Backend Bound (43.2% on Atom): The Efficiency cores are struggling with the backend, likely waiting for data to arrive from memory.

Question 2: Matrix Multiply

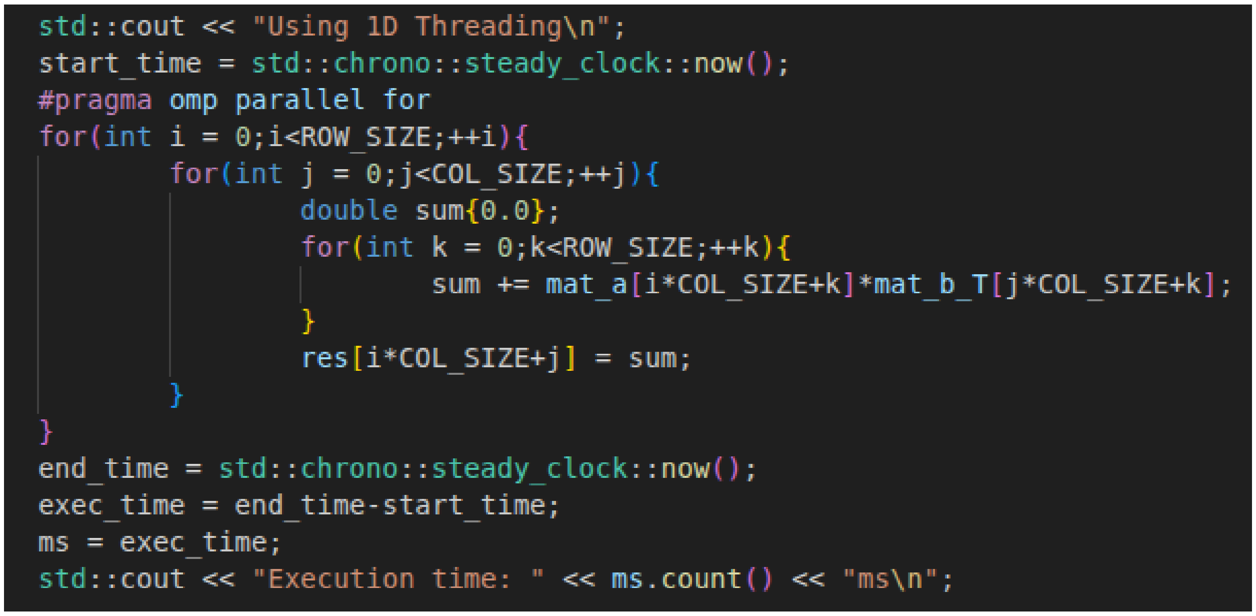
Using Code:-

Using No Threading:



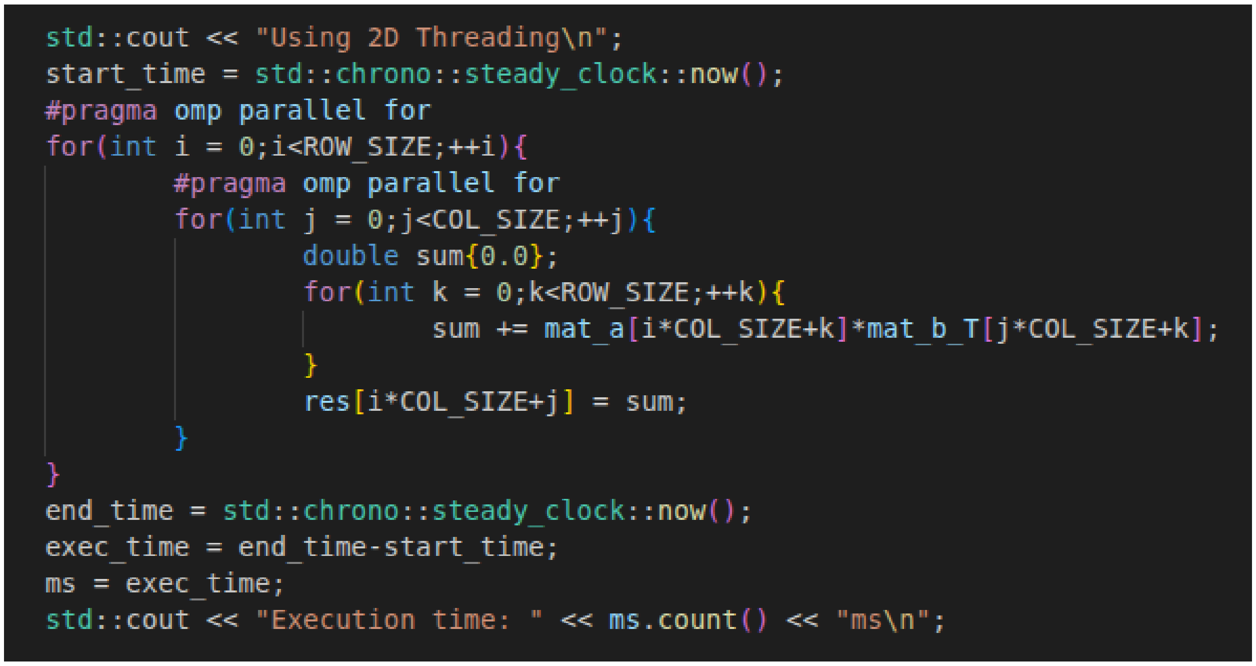
```
std::cout << "Using No Threading\n";
auto start_time{std::chrono::steady_clock::now()};
for(int i = 0;i<ROW_SIZE;++i){
        for(int j = 0;j<COL_SIZE;++j){
                double sum{0.0};
                for(int k = 0;k<ROW_SIZE;++k){
                        sum += mat_a[i*COL_SIZE+k]*mat_b_T[j*COL_SIZE+k];
                }
                res[i*COL_SIZE+j] = sum;
        }
}
auto end_time{std::chrono::steady_clock::now()};
auto exec_time{end_time-start_time};
std::chrono::duration<double, std::milli> ms{exec_time};
std::cout << "Execution time: " << ms.count() << "ms\n";
```

Using 1D Threading:



```
std::cout << "Using 1D Threading\n";
start_time = std::chrono::steady_clock::now();
#pragma omp parallel for
for(int i = 0;i<ROW_SIZE;++i){
        for(int j = 0;j<COL_SIZE;++j){
                double sum{0.0};
                for(int k = 0;k<ROW_SIZE;++k){
                        sum += mat_a[i*COL_SIZE+k]*mat_b_T[j*COL_SIZE+k];
                }
                res[i*COL_SIZE+j] = sum;
        }
}
end_time = std::chrono::steady_clock::now();
exec_time = end_time-start_time;
ms = exec_time;
std::cout << "Execution time: " << ms.count() << "ms\n";
```

Using 2D Threading:



```
std::cout << "Using 2D Threading\n";
start_time = std::chrono::steady_clock::now();
#pragma omp parallel for
for(int i = 0;i<ROW_SIZE;++i){
        #pragma omp parallel for
        for(int j = 0;j<COL_SIZE;++j){
                double sum{0.0};
                for(int k = 0;k<ROW_SIZE;++k){
                        sum += mat_a[i*COL_SIZE+k]*mat_b_T[j*COL_SIZE+k];
                }
                res[i*COL_SIZE+j] = sum;
        }
}
end_time = std::chrono::steady_clock::now();
exec_time = end_time-start_time;
ms = exec_time;
std::cout << "Execution time: " << ms.count() << "ms\n";
```

ON EXECUTION:

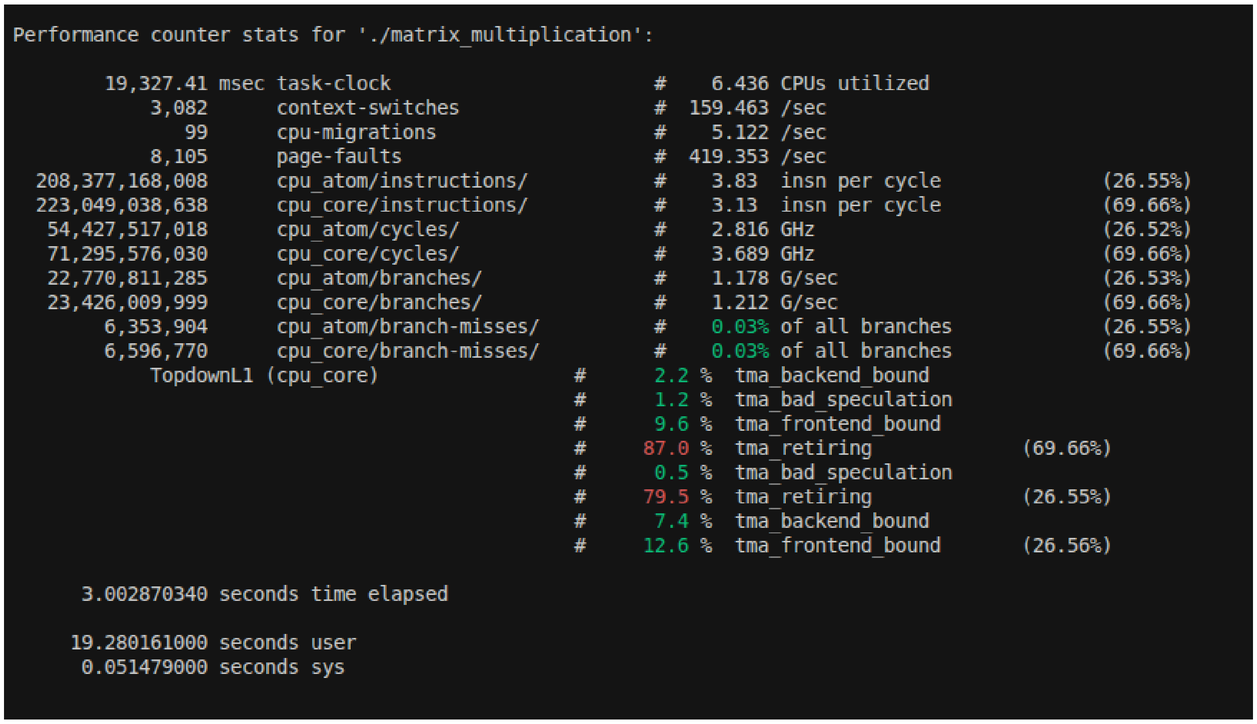
Speedup = sequential time / parallel time

Strategy	**Execution Time (ms)**	**Speedup Factor**
Serial	2557.42	1x
Parallel 1D	620.913	4.12x
Parallel 2D	603.341	4.24x

INFERENCE:-

The analysis of the execution times, as presented in the table, reveals a substantial reduction in time when transitioning from serial to parallel execution. This transition yields an approximate fourfold speedup. This significant acceleration is attributed to the fact that the threaded, parallel implementation utilizes the CPU resources more effectively than the sequential method, which was underutilizing the CPU. Furthermore, the incorporation of 2D threading resulted in an additional decrease in execution time, thereby contributing to a marginally improved overall speedup.

OBSERVATION FROM PERF STAT:



```
Performance counter stats for './matrix_multiplication':

        19,327.41 msec task-clock                       #    6.436 CPUs utilized
            3,082      context-switches                 #  159.463 /sec
               99      cpu-migrations                   #    5.122 /sec
            8,105      page-faults                      #  419.353 /sec
  208,377,168,008      cpu_atom/instructions/           #    3.83  insn per cycle              (26.55%)
  223,049,038,638      cpu_core/instructions/           #    3.13  insn per cycle              (69.66%)
   54,427,517,018      cpu_atom/cycles/                 #    2.816 GHz                         (26.52%)
   71,295,576,030      cpu_core/cycles/                 #    3.689 GHz                         (69.66%)
   22,770,811,285      cpu_atom/branches/               #    1.178 G/sec                       (26.53%)
   23,426,009,999      cpu_core/branches/               #    1.212 G/sec                       (69.66%)
        6,353,904      cpu_atom/branch-misses/          #    0.03% of all branches             (26.55%)
        6,596,770      cpu_core/branch-misses/          #    0.03% of all branches             (69.66%)
            TopdownL1 (cpu_core)                 #      2.2 %  tma_backend_bound
                                                 #      1.2 %  tma_bad_speculation
                                                 #      9.6 %  tma_frontend_bound
                                                 #     87.0 %  tma_retiring          (69.66%)
                                                 #      0.5 %  tma_bad_speculation
                                                 #     79.5 %  tma_retiring          (26.55%)
                                                 #      7.4 %  tma_backend_bound
                                                 #     12.6 %  tma_frontend_bound    (26.56%)

      3.002870340 seconds time elapsed

     19.280161000 seconds user
      0.051479000 seconds sys
```

1. Excellent Parallel Utilization

Metric: 6.436 CPUs utilized

Insight: Your program maintained an average of approximately 6.5 busy cores simultaneously. This high utilization confirms that your parallel code is successfully and effectively distributing the intense mathematical workload across your available hardware.

2. Superior Computational Efficiency (High IPC)

Metric: Instructions Per Cycle (IPC) of 3.13 on main cores

What This Means: The CPU is completing more than three instructions for every clock cycle.

Benefit: This exceptional IPC value is characteristic of a math-heavy workload like matrix multiplication. It indicates the CPU is dominantly spending its time on actual calculations, rather than being stalled and waiting for data to arrive from the RAM.

3. Near-Perfect Logic Prediction

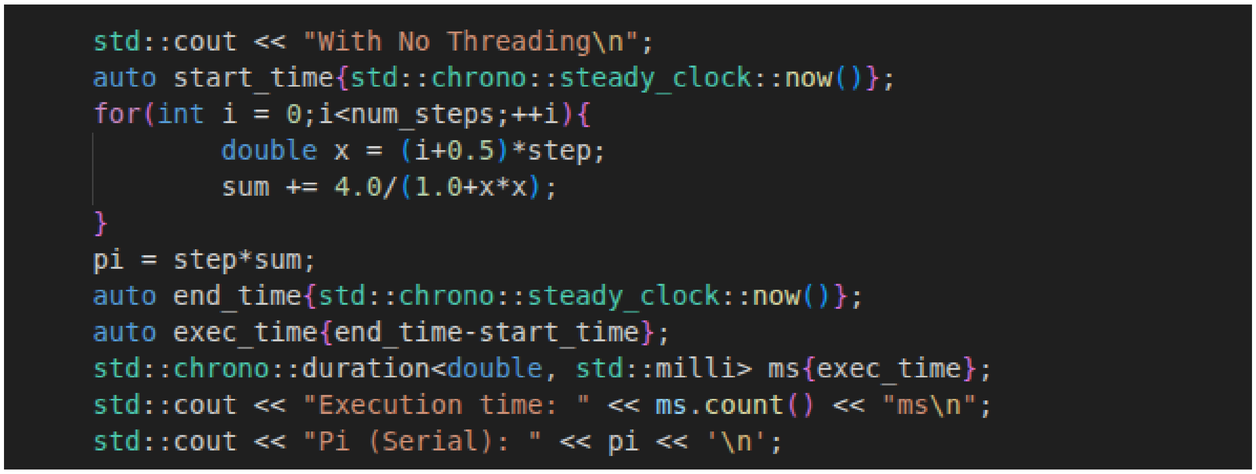
Metric: Extremely low branch-miss rate of only 0.03%

Key Takeaway: The CPU is correctly anticipating the program's next step an astonishing 9,997 out of 10,000 times. The simple, repetitive structure of matrix multiplication loops allows the hardware's predictive logic to almost flawlessly stay ahead of the computational demands.

Question 3: Calculation of π

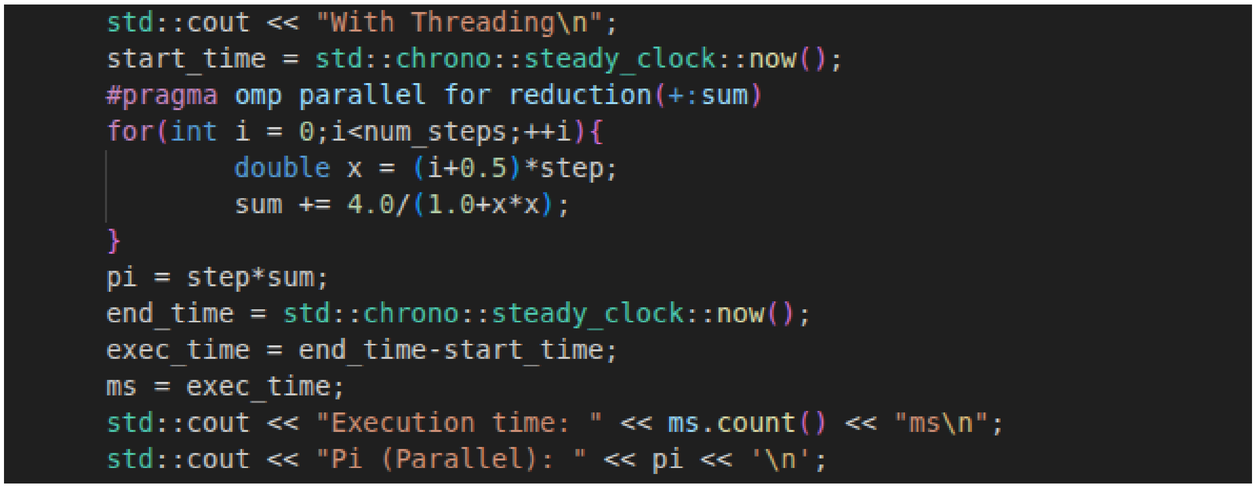
Using Code:

Without Threading:



```
std::cout << "With No Threading\n";
auto start_time{std::chrono::steady_clock::now()};
for(int i = 0;i<num_steps;++i){
        double x = (i+0.5)*step;
        sum += 4.0/(1.0+x*x);
}
pi = step*sum;
auto end_time{std::chrono::steady_clock::now()};
auto exec_time{end_time-start_time};
std::chrono::duration<double, std::milli> ms{exec_time};
std::cout << "Execution time: " << ms.count() << "ms\n";
std::cout << "Pi (Serial): " << pi << '\n';
```

With Threading:



```
std::cout << "With Threading\n";
start_time = std::chrono::steady_clock::now();
#pragma omp parallel for reduction(+:sum)
for(int i = 0;i<num_steps;++i){
        double x = (i+0.5)*step;
        sum += 4.0/(1.0+x*x);
}
pi = step*sum;
end_time = std::chrono::steady_clock::now();
exec_time = end_time-start_time;
ms = exec_time;
std::cout << "Execution time: " << ms.count() << "ms\n";
std::cout << "Pi (Parallel): " << pi << '\n';
```

In this program the threads have to cooperate with each other for computation

ON EXECUTION:

Speedup = sequential time / parallel time

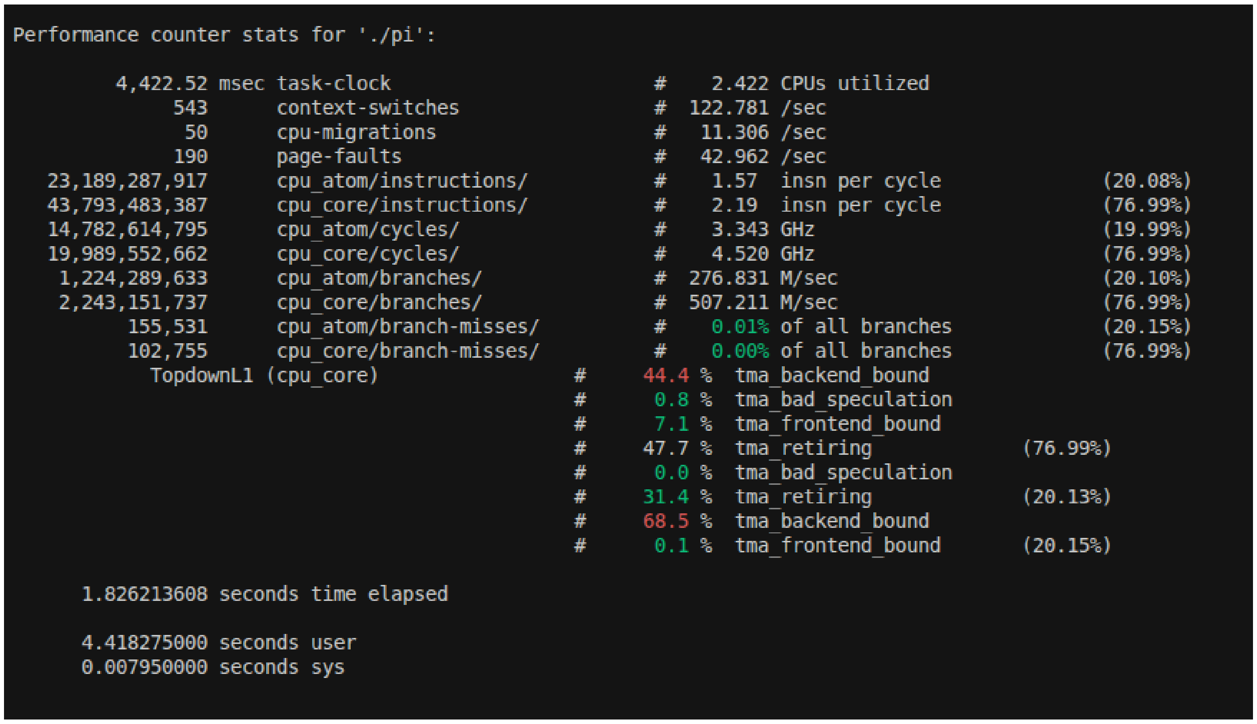
Strategy	**Execution Time (ms)**	**Speedup Factor**
Serial	1634.86	1x
Parallel 1D	175.447	9.32x

INFERENCE:

Both methods yielded the same value for pi: 3.14159. The transition to parallel execution resulted in a significant performance improvement, drastically reducing the execution time and achieving a speedup factor of approximately 9x. This substantial speedup suggests a much more efficient utilization of the CPU in the parallel implementation.

The accuracy of the result confirms the correct cooperation between threads and the successful elimination of the race condition, which is a common concern in problems where threads are mutually dependent. This outcome demonstrates the robust and superior performance of parallel execution, even in programs susceptible to race conditions.

OBSERVATION FROM PERF STAT:



```
Performance counter stats for './pi':

         4,422.52 msec task-clock                       #    2.422 CPUs utilized
              543      context-switches                 #  122.781 /sec
               50      cpu-migrations                   #   11.306 /sec
              190      page-faults                      #   42.962 /sec
   23,189,287,917      cpu_atom/instructions/           #    1.57  insn per cycle              (20.08%)
   43,793,483,387      cpu_core/instructions/           #    2.19  insn per cycle              (76.99%)
   14,782,614,795      cpu_atom/cycles/                 #    3.343 GHz                         (19.99%)
   19,989,552,662      cpu_core/cycles/                 #    4.520 GHz                         (76.99%)
    1,224,289,633      cpu_atom/branches/               #  276.831 M/sec                       (20.10%)
    2,243,151,737      cpu_core/branches/               #  507.211 M/sec                       (76.99%)
          155,531      cpu_atom/branch-misses/          #    0.01% of all branches             (20.15%)
          102,755      cpu_core/branch-misses/          #    0.00% of all branches             (76.99%)
            TopdownL1 (cpu_core)                 #     44.4 %  tma_backend_bound
                                                 #      0.8 %  tma_bad_speculation
                                                 #      7.1 %  tma_frontend_bound
                                                 #     47.7 %  tma_retiring             (76.99%)
                                                 #      0.0 %  tma_bad_speculation
                                                 #     31.4 %  tma_retiring             (20.13%)
                                                 #     68.5 %  tma_backend_bound
                                                 #      0.1 %  tma_frontend_bound       (20.15%)

      1.826213608 seconds time elapsed

      4.418275000 seconds user
      0.007950000 seconds sys
```

1. Threading and Coordination

The program demonstrates a Moderate Parallel Effort with 2.422 CPUs utilized, maintaining roughly 2.5 active cores on average.

The Observation: While multithreading is used, the processor isn't fully saturated, unlike the matrix multiplication example. This lower utilization is typical for the Pi problem because threads must spend time coordinating to merge their results into the final value, introducing a correlation overhead.

2. Strong Computational Efficiency

The system achieves a healthy IPC (Instructions Per Cycle) of 2.19 on the performance cores.

The Insight: This is a very efficient number, indicating the CPU successfully dedicates its time to the actual math of the integral (calculating rectangle areas) rather than wasting cycles waiting for data.

3. Optimized Code Flow

The code exhibits Near-Perfect Branch Prediction, with the branch-miss rate on performance cores essentially 0.00%.

The Significance: Because the code's logic is highly repetitive and straightforward, the CPU correctly predicts every subsequent step. This eliminates "Bad Speculation" (wasted work), which accounts for a tiny 0.8% of total execution time.